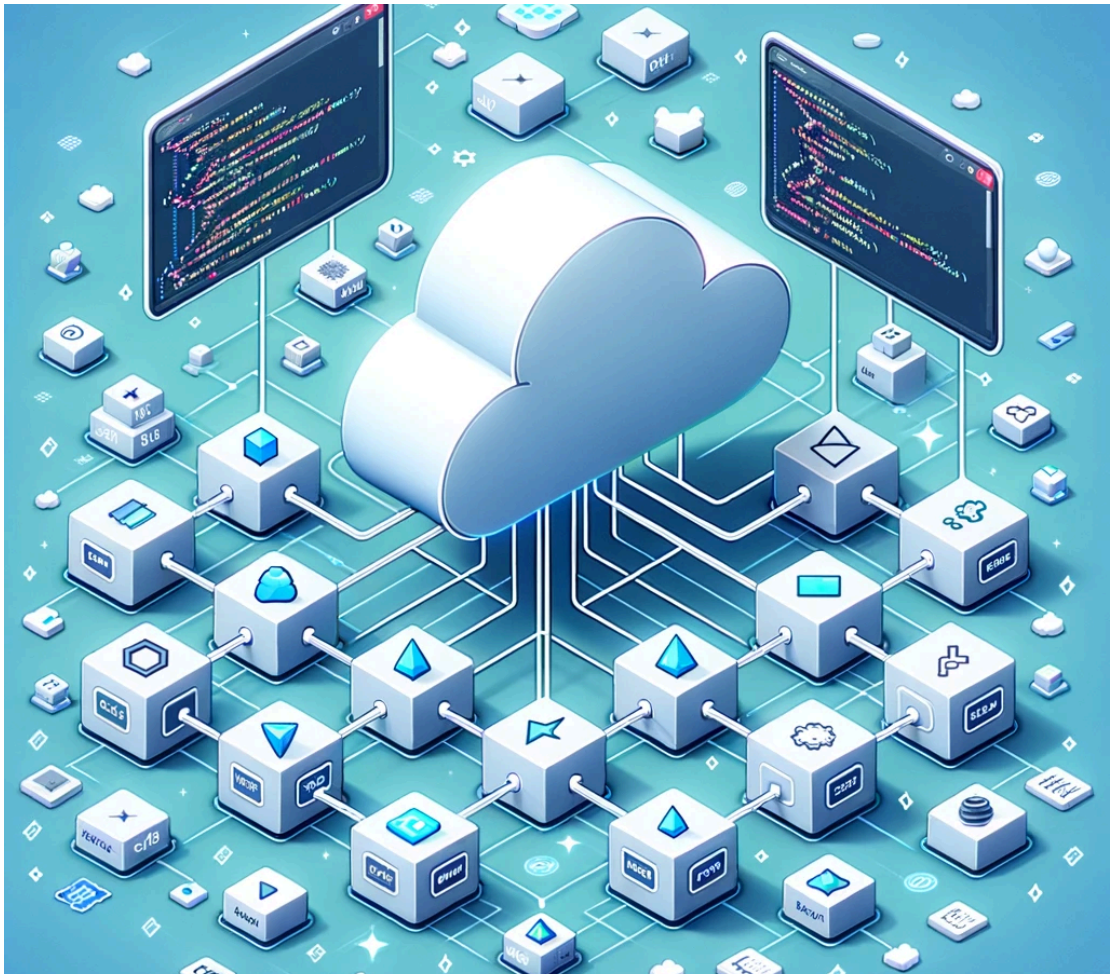
Our Journey to Cloud Native architecture for Low Latency trading systems



What is Cloud Native Architecture?

Over the past decade there has been an overall IT move to the cloud, and best practice guidelines have emerged (such as the "The 15 Facor App"), to help people build scalable, cloud native applications and solutions. Within the trading and exchange environment, this adoption has been tempered by the requirement for very low latency, because highly scalable web technologies are not generally low latency. For example, Kafka - a well known cloud messaging technology - is great for operating at scale, but it has very high latency end to end, making it inappropriate for trading environments.

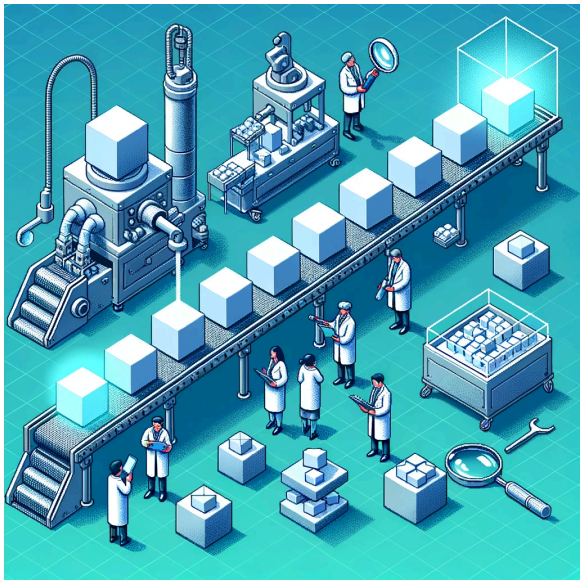
The following describes our journey to our cloud native solution.

Our Journey to the cloud

We recognised the need to rebuild our architecture, when in 2017 and 2018 we used our low latency HFT matching engine as the basis for a full Retail Marketplace solution. This required the complete functionality of a retail facing exchange, including such retail components such as payment gateways, KYC onboarding and referral programs. Amongst all the hype, we faced potential issues such as live KYC account opening requests numbering in the thousands per hour, and the need for elastically scalable front end servers to feed price information out to tens of thousands of users.

As a result,from 2019, and for a period of over 3 years, we redefined and rebuilt our architectural approach. We could not make a simple jump to adopt the latest approaches, but instead needed a more nuanced approach.

We clearly had to maintain our market leading latency, to serve the HFT clients in Japan, but we needed to combine this with the best of the "Cloud Native" technologies to allow us to also serve retail markets and our scalability, flexibility goals.



This meant that we needed to immerse ourselves in "Cloud Native"technologies, to understand the trade-offs first. Use research and prototyping to get intimate with microservices, micro-front ends, containers, orchestration and deployment models, messaging systems. Then pull back to make tough decisions on whether to use our existing approaches, or rebuild to adopt the cloud approaches.

It was a huge risk for a company of our size, but we knew that it was time to take the plunge. Cards up in the air.

The following details some of the choices we settled on.

The Challenge and Our Solution

To meet the needs of our latency-sensitive clients, we've re-engineered our solutions from the ground up, cherry picking from cloud native architectures as much as possible to achieve a solution that is innovative and optimized for the needs of our clients. We are proud of the work we have done, and would like to share our experiences and the decisions and work we undertook.

Scalability and Performance

We moved our core processing engines from shared memory engines (similar to the Fidessa model) to a fully "Event Source" streaming model, where any subscriber compoent can be instantly replaced by another. This" streaming" model is one of the most significant and challenging changes, and this is what gives the "Cloud" its superior reliability to traditional approaches.

For latency on the latency critical paths, it is critical to get the queuing, threading, logging and memory management well researched and optimized. Meanwhile, for scale, sharding, fan out and stateless components are necessary parts of the messaging. For both of these we chose to build these in-house, as we did not feel the available cloud solutions would meet our requirements.

Built out several high speed "Micro-service Aggregates" as query handlers in the new event source based Command Query Responsibility Separation (CQRS) model, to support dramatically increased scale, reduced "single point of failure", and dramatically improved latency for aggregate data requests such as tick data (typically 5 to 10 ms).

Breaking down the monolith

While we already had a distributed service model, we did previously have a single large database - known as a monolithic database. This had grown to be a bottleneck - both for scaling customers, and for scaling our development teams. It was simply too large for anyone but the original senior developers to understand. , and a and shifted to simpler, smaller databases, tuned to the requirements of each service.

As part of the move to micro-services, and also driven by two customers requesting a move to lower cost open source databases, we broke down our single, large monolithic database with thousands of tables (which was mostly managed by the support team and Business domain experts) to much smaller databases - one for each microservice - which are defined and managed by the development team.

This single move had a large positive impact on our development teams, as it suddenly became much easier to understand the full scope of each database, and thus empowered to discuss improvements.

At the same time, with the move to having identical databases being deployed in multiple customers (instead of custom databases for each customer), we also moved to using "Migration" tools to manage database design updates deployed in the same automated way as code updates.

It was therefore required to differentiate between common services and data models which do not vary between customer deployments, and the highly customized services for integrations that many customers need. These two initiatives dramatically improved our ability to support more consistency between customer deployments, while still achieving our core mission of providing custom solutions.

With the “single” database model gone, we were able to finally drop our dependence on expensive legacy database systems,

An in-memory distributed database, for keeping metadata updated in real-time in the latency sensitive services, such as the quote pricing engine, the unified matching engine, our hedging engine, and risk management engine and to provide external mapping data to all our integration adapters (such as our FIX connections to Intertrade, Tora, PrimeXM, BlockFill, the JPX and TFX, and connections to Metatrader). As a young service company, we always assumed that we would need to provide mapping both inbound and outbound!

A postgresql database for configuration management, transaction management, archive data and report handling, which is still able to support our “multi-tenant” model.

An influxDB time series database to provide the backing store for our real-time OHLC bar factory, and optional Integrations with external cloud data lake services (required for our work with Barclays).

Microservices and System Architecture

We adopted a selective microservice model, applying a "Domain Driven Design" approach with "bounded contexts" to split out core functionality, while ensuring that low latency stateful components remain extremely fast.

We adopted Event Sourcing, with a new low latency messaging tier, micro-service architecture, and common Service Mesh and Service Chassis framework for service discovery

We rebuilt our Service Orchestration functionality based on a common Service Mesh and automate containerised deployments.

We underwent a major database and data model transition from a large monolithic database design (which was limiting the team's ability to innovate), to a microservice based database design, which can now operate on multiple databases, including postgresql.

Interoperability and Standards

The architecture is based on the MS Open Application Model (OAM), a cloud first design, allowing us to manage multiple environments from a single model. Consistent deployments models to dev, UAT or production environments - while still supporting the customer approval model required in our regulated customers.

Service Mesh and Service Chassis “Framework”

Our service mesh allows any service in the system to locate, monitor or communicate with any other service via the service mesh. Meanwhile, the Service Chassis provides common system wide functionality, such as distributed real-time data (Venue, Instrument and Trade Account data), transaction message routing and event sourcing, observability, and process scheduling functionality.

This common application framework dramatically accelerates the development process when creating new solutions or adding components to a system.

Development Workflow and Code Quality

We migrated to Gitlab for source code control, and to empower a significant "Shift-Left" transition in our testing approach, and guaranteed code review/approval process.

While not strictly an architecture issue, at the same time as building out the new architecture, we applied multiple modern approaches to our development lifecycle. In a move towards fully automated "Shift Left" testing, we required that all components can be fully tested by the developer, using mock everything. We built mock message injectors for our Itch, Glimpse and old OMX exchange feeds, for our FIX connections to counterparties like B2C2, Bloomberg, CQG and Trading Technologies. Databases, Detadata, Telemetry, Schedule providers. Everything.

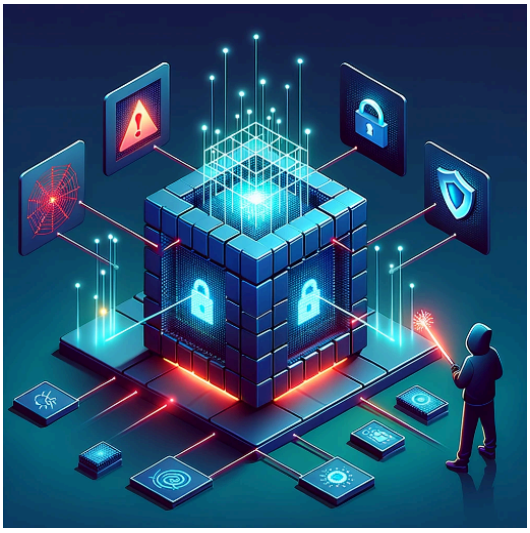
We built a brand new, unified Behaviour Driven Design (BDD) test framework called "TFCoverage", a multi-session environment builder and end-to-end tester, which can operate simultaneously across multiple FIX sessions, Websockets, REST sessions and UI sessions. This generates our test environments and runs the end to end tests as part of every build cycle.

We built SAST and DAST code analysis into our Continuous Integration Pipelines, to reveal code weaknesses before being deployed to test environments.

Structured version controlled dependency management, allowing customer releases to be matched with the specific dependencies required.

Security and Compliance

Security Management is deeply integrated into our operations, including a regular Risk Management process periodic accredited external penetration tests and DAST/SAST automated tests built into our build pipeline.



Security and Access Control

With the move to the microservice architecture, we completely rebuilt our security implementation to support

distributed components, while retaining our existing time proven multi-layered account data model, which has proven so flexible to numerous different customer implementations, from exchanges to retail traders.

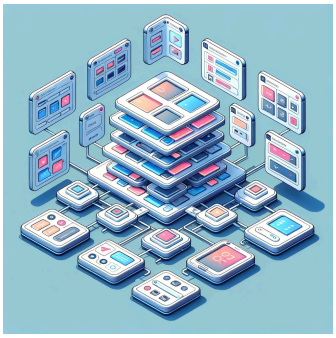
A new GraphQL based back end service integrated to the new granular user role based security model, covering both row level access control and feature provisioning

Our alignment with ISO 9000 and FSA guidelines (with annual audits) ensures that your data is secure and compliant.

Frontend and User Experience

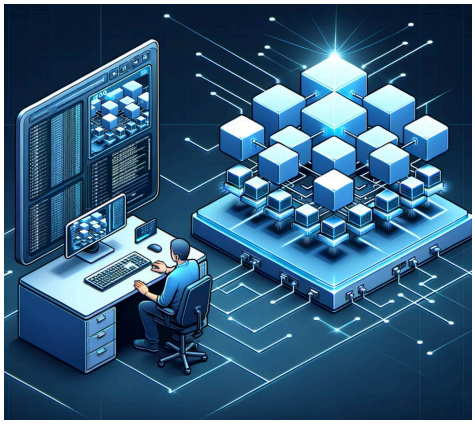
We dropped the front end legacy framework that was based on Sencha extJS, and moved to ReactJS for all new projects

Built a new common front end chassis including features such as the security and permissioning model, the multi-language and multi-timezone capabilities into the new React.



We rebuilt our front ends based on a micro-front end design, following our existing configurable and granular feature and access permissioning model - leading towards the FDC3 "Financial desktop" standard (still a work in progress). The application lancher as part of this is built with the web standard for re-usable html components, "Web Components", and is therefore front end framework agnostic.

At the same time, we integrated an all new front end grid component system, supporting dramatically higher load and volume.



Observability and System Monitoring

We build a new system monitor system based on a digital twin approach to observability, using the OpenTelemetry standards, our service mesh model, and self registration. We then added both application service controls and integrated docker controls to the same system management console. We also then integrated logstash and Elastic to round off allowing all support operations to be managed from one place.

With the "digital twin" model in place, this could then be used to power cluster management, and instant failover of stateful high availability services.

Time Management and Scheduling

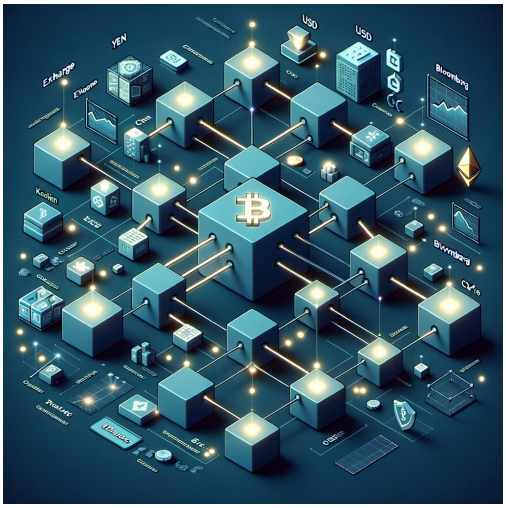
We rebuilt our existing date management and service scheduling functionality to support multiple calendars (exchanges operating on different calendars), multiple time zones (eg. the Tokyo Financial Exchange which operates in Japan on Chicago timezones), multiple session types - daily sessions (futures markets), intra-day sessions (equity markets), week-long sessions, and "Time-Travel" to support both calendar and weekend exchange testing set in the future, and process re-runs and corrections in the past.

We rebuilt our audit and time-travel handling to operate with the latest temporal database designs, while spitting out our archive data service as a real-time listener on the event streams, thus removing the need for a separate archive handling process at the end of every day.

Batch to stream

We have always had an extremely flexible batch processing engine, which easily supports end of day processing and imports and Exports to different external systems. However, this is now being replaced with a stream processing engine, which will support much higher volume, and support system wide account status events, like margin warning and limit kill switch events, for HFT traders.

Connectivity and Business Logic



We were finally ready to integrate (and port of over) significant functionality to the new model, including multiple market makers and exchange gateways (TFX, JPX/Tocom), client FIX gateways, the cross currency matching and pricing engine, our Itch and Glimpse exchange feeds, and our Tora equities connections.

Summary

Ok, so perhaps this is somewhat of a list of features, or perhaps it is a presentation of how much effort goes into bringing a legacy system up to date. At the end of the day, it has been a huge learning experience which is worth sharing.

We now have a much more engaged team, a product which has many many more automated tests built into the release process, and a devops team who are able to automate much of the environment management and to automate much of the monitoring, and are confident enough to consider moves to adding AI to the system monitoring etc.

Of course, it never stops. This is just one chapter in an ongoing process to keep up with customer expectations.